Transparent Result Caching

Amin Vahdat
Computer Science Division
University of California, Berkeley

Thomas Anderson
Department of Computer Science and Engineering
University of Washington, Seattle

Abstract

The goal of this work is to develop a general framework for transparently managing the interactions and dependencies among input files, development tools, and output files. By unobtrusively monitoring the execution of unmodified programs, we are able to track *process lineage*—each process's parent, children, input files, and output files, and *file dependency*—for each file, the sequence of operations and the set of input files used to create the file. We use this information to implement Transparent Result Caching (TREC) and describe how TREC is used to build a number of useful user utilities. *Unmake* allows users to query TREC for file lineage information, including the full sequence of programs executed to create a particular output file. *Transparent Make* uses TREC to automatically generate dependency information by observing program execution, freeing end users from the need to explicitly specify dependency information (i.e., Makefiles can be replaced by shell scripts). *Dynamic Web Object Caching* allows for the caching of certain dynamically generated web pages, improving server performance and client latency.

1 Introduction

The goal of this work is to develop a general framework for transparently managing the interactions and dependencies among input files, development tools, and output files. By unobtrusively monitoring the execution of unmodified programs, we are able to track *process lineage*—each process's parent, children, input files, and output files, and *file dependency*—for each file, the sequence of operations and the set of input files used to create the file. This information can be used to determine the exact sequence of operations used to create any system file or to keep the contents of output files synchronized as dependent input files are modified.

As a motivating example, several years ago, it was discovered that some published satellite data had been incorrectly normalized; however, because of the lack of software support, it has been difficult to identify exactly which experimental results were tainted by the error. It is believed that several journal articles have since been published still based on the incorrect data [Dozier 1993]. As another example, one error common to program developers is introducing a new header file without manually updating dependency information. This can result in an executable with object files based on different versions of the same header file, often resulting in subtle bugs where different modules reading and writing the same fields of a data structure in fact access different regions of memory.

The combination of transparently obtaining process lineage and file dependency information provides a powerful substrate for developing applications in a wide range of application domains.

- *Unmake:* Unmake allows users to query TREC for file lineage information, including the full sequence of processes executed to create a particular output file. For example, users running simulations that neglect to document the parameters to generate output files can query Unmake for the program used to create the output, the specific command line parameters, and the environment variables in effect when the command was executed.
- *Accountability and Access Control:* Related to the Unmake application, TREC can be used to perform logging of programs run as `root`. System administrators are often forced to give `root` privileges to multiple users. Unfortunately, this means that accountability is sacrificed, making it difficult to ascertain the identity of individuals responsible for particular actions (e.g., removing an important file). Since Unmake can provide process

lineage information, it can trace file accesses back to the shell (and hence the user id) of the process that originally executed `su`. Such a tool can also be extended to monitor system calls, disallowing certain accesses based on the "effective" `uid` of the calling process [Goldberg et al. 1996]. For example, Bob acting as root may be disallowed write access to all files in `/dev` and `/etc`.

- *Transparent Make:* This version of the `make` utility allows users to specify the sequence of operations for constructing output files as simple shell scripts. The first time the shell script is run, TREC determines the set of files that affect output files through empirical observation. During subsequent executions of the shell script, TREC can re-run only those commands that have been invalidated by changes to input files. This approach has two principal advantages. First, it frees users from manually specifying dependency information in a language than can be restrictive [Levin & McJones 1993]. Next, transparent make does not require users to manually update dependency information. Thus, when a new header file is added to a source tree, TREC transparently adds the new dependency to its lineage information by observing the inputs to subsequent compilations. Similarly, if a tool's command line parameters must continuously be updated to produce output files, TREC automatically matches each output file to the parameters used to generate it.

- *Dynamic Web Object Caching:* Today many web pages are constructed dynamically as a result of user input. One example in the web today is using CGI-bin programs to produce HTML pages. For instance, to download the latest version of Navigator, users answer a number of questions about their platform before being presented with a page enumerating URLs for the correct binary. The disadvantage of using CGI-bin programs is that a web server must generally `fork`, then `exec` a program to produce the HTML content. Given some locality in user input, it would be cheaper to cache the results of CGI-bin program execution with popular input patterns (e.g., users wishing to download the latest English version of Navigator for Windows95/NT), reducing both server load and client latency. TREC allows for such caching with invalidations to handle the case where input to a CGI program changes (e.g., a new version of navigator becomes available). This application is more active than the previous two examples: TREC dependency information is used to generate specific actions whenever a pre-specified operation takes place (an output file is invalidated when its input files are modified).

In this paper, we demonstrate how our prototype framework for *Transparent Result Caching* (TREC) is used to implement three of the above applications: unmake, transparent make, and dynamic web object caching.

Currently, `make` and related software configuration management tools are used for specifying and maintaining dependency information. Such tools suffer from a number of deficiencies. For example, to manage file and program dependencies, users must manually specify dependency information. Programmers must specify the dependencies between source files, object files, and executables. If any changes occur, such as introducing a new header file, users must remember to manually update the dependency information. With TREC, maintaining dependency information is both simpler and less error-prone because dependencies are deduced transparently by observing program execution.

Another shortcoming of `make` and related tools is the inability to track file lineage. Makefiles only implicitly contain lineage information; if the Makefile changes, the lineage information about existing output files can also be destroyed. As described above, lineage information is helpful in a number of contexts. If an output file does not contain expected results, debugging is easiest by working backwards to see whether the problem is with the file inputs, the data analysis tool, or the command line parameters. Similarly, if an input file is discovered to have a flaw, it is helpful to know all the output files derived from the input.

Finally, `make` is largely targeted toward software development; it can be too static to be useful for other communities. For instance, scientists often spend their time exploring different sequences of tools, different parameters, and different parts of an image. For example, one tool might extract the pixel values for a latitude and longitude region from a set of files containing satellite images. However, the images can overlap, and the requested region may span multiple image files. The input files actually read to create an output file can vary depending on the command line parameters passed to the tool. Expressing such dynamic dependencies can be difficult with Makefiles. TREC, on the other hand, is well-suited for managing dynamic dependencies because of its ability to discern file lineage simply by observing program execution.

Our approach to transparently capture file dependency information is to intercept a small number of system calls using native kernel tracing mechanisms. Using the information from these calls, TREC maintains the following information for each process: command line arguments, environment variables, process parent, process children, files read (input files), and files written (output files). TREC then organizes this information hierarchically, allowing users to query the system for file lineage, for example, to determine all processes involved in creating an output file.

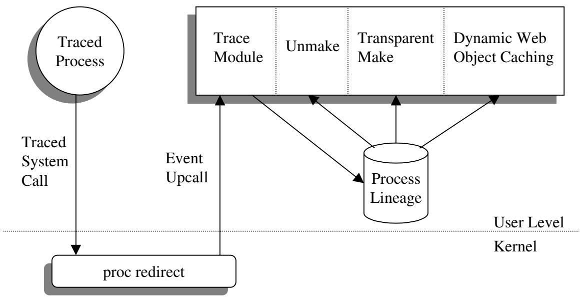


Figure 1: This figure describes the TREC architecture.

With transparent make for example, we are able to automatically determine the set of include files associated with any compilation by observing the I/O behavior of the compiler. Thus, when an include file is modified, transparent make automatically determines that re-compilation is necessary from process lineage information. Relative to existing techniques for automatically determining dependencies, our approach has the added benefit of not requiring a separate parser for each different programming language syntax.

Our initial implementation of TREC uses the `proc` file system facilities of Solaris to intercept the necessary set of system calls. Our implementation of TREC, its baseline performance, and limitations of TREC profiling are described in detail in Section 2. In Section 3, we describe how TREC is used to implement our three sample applications, unmake, transparent make, and dynamic web object caching. Section 4 describes related work and Section 5 presents our conclusions.

2 Implementation

2.1 Architecture

TREC is implemented using the Solaris `proc` file system (most major UNIX variants provide a substantially similar interface) to intercept the set of system calls necessary to build dependency information. For our target UNIX architecture this set includes the following system calls: `open`, `fork`, `fork1`, `creat`, `unlink`, `exec`, `execve`, and `exit`. By catching these system calls, TREC is able to determine full process lineage information. Command line parameters and environment variables are available from the `exec` system calls. TREC determines the set of input files and output files by examining the options to the `open` system call (targets of `creat` are assumed to be output files). This design choice potentially over-constrains the set of input and output files because, for example, a file opened for writing that is not actually written to with a `write` system call is considered an output file. We made this design decision because of the added overhead of intercepting all `read` and `write` system calls in addition to the set of calls already being monitored. For some programs, we observed that `read` and `write` operations were executed four times as often as all traced system calls combined.

The overall TREC system architecture is summarized in Figure 1. TREC runs as a multi-threaded process that attaches to a target process using the `proc` file system interface. A trace thread is responsible for building lineage information. Other threads use this lineage information to implement the higher-level services described in Section 3. Given a list of target system calls, `proc` forces a context switch to the TREC tracing thread whenever a relevant call is executed by the attached process or any of its children. The tracing module exports a callback-based interface to application modules. Currently, callbacks are exported for file read and write events. Modules, such as transparent make, use the callbacks to determine when output files are invalidated. Thus, when a callback is received that a file is modified, a module can take action on dependent output files—invalidating the output or re-generating it for example.

TREC uses the system call information to build a lineage tree of the target process and all of its children. Each node of the lineage tree represents an executed process and contains the following information: execution time, command line arguments, environment variables, files read, and files written. An example of this lineage information is presented in Section 3.1.

Operation	Baseline	Traced	Syscall Rate	Added Overhead
`open` syscall	12.4 s	19.3 s	403 calls/s	54.8%
Compile	128.4 s	146.2 s	160 calls/s	13.9%
Latex	35.1 s	36.3 s	16 calls/s	3.5%

Table 1: This table describes the overhead introduced by adding TREC profiling.

2.2 Performance

Since the context switches imposed by the `proc` file system required to perform our tracing can impose significant overhead, we took a number of measurements to quantify the slowdown. Table 1 quantifies the TREC overhead for three simple benchmarks. All benchmarks were conducted on a Sun Ultra/1 workstation running Solaris 2.5.1. The first, open, calls `open` and `close` on the same file in a tight loop 5000 times (note that only the open call is actually traced). While the 54.8% overhead imposed by TREC is significant, the next two benchmarks demonstrate the slowdown of individual system calls do not adversely affect the performance of real applications. The next benchmark is a compilation of the Apache HTTP server, version 1.2.4 [Apa 1995]. The source tree consists of 38,000 lines of C code and was compiled over NFS. While the 13.9% slowdown is noticeable, we believe it to be tolerable. The final benchmark, Latex, involved running `latex` four times, `bibtex`, and finally `dvips` to produce postscript for a 17 page document. For this benchmark, only a 3.5% overhead is introduced. As indicated by the "Syscall Rate" column in Table 1, the measured slowdown directly corresponds to the rate at which processes execute traced system calls. Since the Latex benchmark executes only 16 traced system calls per second, it suffers the smallest slowdown.

To address the overhead imposed by the `proc` and related tracing facilities, we could implement TREC functionality in the kernel. Various tools such as Watchdogs [Bershad & Pinkerton 1988], and Interposition Agents [Jones 1993], or SLIC [Ghormley et al. 1996] can be used to trace system call activity with little or no overhead. However, such tools often require root access to install, can be difficult to use without kernel source, and can also be difficult to distribute since kernel copyright restrictions may prevent distribution of source code. We opted for the user-level approach for portability, ease of distribution and installation. If performance becomes an issue, we believe switching to a kernel implementation will be straight-forward.

2.3 Limitations

As motivated earlier, a number of applications are able to benefit from TREC functionality. However, TREC can produce incorrect results for applications that base their results on non-deterministic or difficult-to-trace input. Following are some example behaviors likely to interact poorly with TREC applications:

- Programs must be deterministic and repeatable. Each of the programs that contribute to the creation of a file must behave the same way each time it is run, assuming that its own inputs have not changed. A compiler invoked with a given set of options will generally produce the same object file, as long as the source code has not changed. An example of a program that violates this restriction is UNIX `date`, whose output is never the same twice. Any file that relies on the output of `date` cannot be guaranteed to be up-to-date, nor can it be reliably re-created.

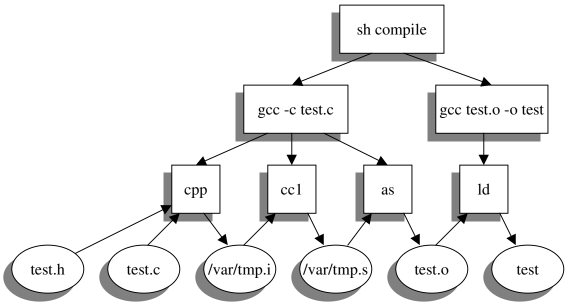
- Programs cannot rely on user input. Related to the requirement for determinism, programs that rely on user input or GUI operations are not automatically re-creatable. For example, if an output file incorporates user-input to a text editor, TREC cannot accurately model program dependencies.

- File contents must be static, as long as the modification time has not changed. While seemingly a trivial constraint, certain special files do not follow this convention. Virtually all of the files in `/dev` violate this restriction. For example, each time the tape drive `/dev/rmt0` is read, it appears to have different data, for example, because an operator has exchanged one tape for another.

- File contents must be changed locally. For example, an NFS mounted file might be modified at any of a number machines, not all of which may be traced by TREC. Applications requiring callbacks on file modification rely on TREC's ability to intercept all file updates.

```
gcc -c test.c
gcc test.o -o test
```

(a) Compile Script



(b) Process Lineage

Figure 2: The top portion of the figure shows a simple shell script used to compile a program, `test`*. The bottom portion of the figure graphically depicts the process lineage tree produced by TREC-profiling of the shell script.*

- In general, TREC cannot cache programs that rely on network communication to produce their output. For example, an application that communicates with a remote server may receive different results for each run of the program.

Despite the limitations outlined above, we will demonstrate that TREC remains a useful tool in a number of different contexts. Our current approach to handling output files produced by programs that fall into the above categories is to allow users to specify a set of program names whose output cannot be cached or re-created. TREC uses a configuration file containing a list of programs whose output is potentially uncacheable. If any of these programs are involved in producing an output file, TREC caching is disabled (i.e., by transparent make or dynamic object caching). In the future, it should be possible to partially automate the process of determining the list of programs displaying "dangerous" behavior. For example, Solaris programs opening `/dev/tcp` can be assumed to be carrying out network communication.

Note that while TREC may be unable to transparently cache the results of certain programs, applications such as unmake can still provide valuable information to users about the origins of even uncacheable files. For example, if an image file is created using an interactive visual analysis tool, the output cannot be transparently re-created since user input was used to drive the result. However, unmake can still be used to identify the command executed to start the analysis tool, to determine the time the command was executed, or to enumerate all input files used during the creation of the image file.

3 Applications

In this section, we describe three utilities built on top of the TREC tracing module: unmake, transparent make, and dynamic web object caching.

3.1 Unmake

The TREC tracing module builds a process lineage tree as described in the previous section. To demonstrate the use and utility of this information, we describe a simple TREC service, unmake, that allows for a number of simple queries. For example, users might request information about all processes, and their parents, that read a particular file.

As a concrete example, consider the shell script used to compile a simple program in Figure 2(a). When this script is run in a shell traced by TREC, the tracing module automatically builds lineage information for the processes executed by the script. The complete process lineage tree for producing the `test` executable is presented in Figure 2(b). Arrows between rectangles indicate process parent information. For the leaf processes, we indicate the files (indicated as ovals) read and written by each process. Notice that while the compilation script makes no reference to a header

Query: read test.c

```
Parent ID: 28426 Program ID: 28428
Argv:   /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/cpp
        -lang-c -undef -D__GNUC__=2 -D__GNUC_MINOR__=7 -Dsun -Dsparc
        -Dunix -D__svr4__ -D__SVR4 -D__GCC_NEW_VARARGS__ -D__sun__
        -D__sparc__ -D__unix__ -D__svr4__ -D__SVR4 -D__GCC_NEW_VARARGS__
        -D__sun -D__sparc -D__unix -Asystem(unix) -Asystem(svr4)
        -Acpu(sparc) -Amachine(sparc) test.c /var/tmp/cca006u2.i
Envp:   COLLECT_GCC=gcc HOME=/homes/rivers/vahdat HOST=tolt HOSTNAME=tolt
        HOSTTYPE=sun4 LOGNAME=vahdat MACHTYPE=sparc OSTYPE=solaris
Children: (none)
Input:  test.c test.h /usr/include/sys/feature_tests.h /usr/lib/libc.so.1
        /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/include/stdio.h
        /usr/lib/libdl.so.1 /usr/platform/SUNW,Ultra-1/lib/libc_psr.so.1
Output: /var/tmp/cca006u2.i
========
Parent ID: 28424 Program ID: 28426
Argv:   gcc -c test.c
Envp:   HOME=/homes/rivers/vahdat HOST=tolt HOSTNAME=tolt HOSTTYPE=sun4
        LOGNAME=vahdat MACHTYPE=sparc OSTYPE=solaris
Children: 28428 28430 28432
Input:  /usr/lib/libc.so.1 /usr/lib/libdl.so.1
        /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/specs
        /usr/platform/SUNW,Ultra-1/lib/libc_psr.so.1
Output: (none)
```

Figure 3: This figure describes the results of a sample query, tracing back the lineage of the process that read the file `test.c`.

file (`test.h`), the assembler, or loader, TREC is able to build full lineage information by observing the execution of all processes and sub-processes spawned by the compilation. Notice that while the shell script from Figure 2(a) makes no reference to the file `test.h`, TREC is able to transparently deduce this dependency by observing the fact that cpp read `test.h` for input.

The unmake module can be interactively queried for process lineage. Figure 3 shows partial output of a run for a query requesting lineage information for processes reading `test.c`. Unmake searches the lineage information for records where the set of input files contains `test.c`, in this case returning execution of the C pre-processor, `cpp`. Unmake returns the following information about the execution environment of all traced processes. All command line arguments, in addition to the environment variables, are listed (note that for brevity, the list of environment variables is truncated). All of the program's input and output files are listed along with a unique program ID (currently the UNIX pid) and the ID of the process's parent. The children field specifies all spawned processes (no processes were forked in the case of `cpp`). The query also recursively provides information on the process's parent, `gcc` in this case. While omitted from Figure 3, information on `/bin/sh`, the process which executed the compilation shell script, and `/bin/tcsh`, the root process of the TREC trace, is also returned.

While not currently implemented, unmake combined with a source control system or, more generally, a file system capable of transparently producing older file versions [Heydon et al. 1997] can be used to rollback to earlier versions of output files. For example, users debugging a program executable may use an interactive process lineage visualization tool, similar to the display in Figure 2(b), to identify input files that may have potentially introduced bugs. The user could then roll back to an earlier version of the suspect input file (using an appropriate file system or a source control system), instructing unmake to rebuild the output file with the new set of input files.

3.2 Transparent Make

The process and file lineage information produced by TREC can be used to build a more dynamic version of the traditional `make` utility, called transparent make (tmake). With traditional `make`, users are forced to learn a new language for specifying dependencies between input, intermediate, and output files, a process that can become cumbersome and error-prone for a large development efforts. In contrast, tmake allows users to describe the process for creating output files more naturally; shell scripts (Figure 2(a) provides a simple example) describe the sequence of steps used to create an output file.

Thus, rather than forcing users to manually specify dependencies through Makefiles, TREC is able to dynamically determine dependencies by observing the execution of shell scripts. This approach has the following advantages: (i) eliminating user errors that may occur in specifying dependency information, (ii) dynamically updating dependency information as it changes, and (iii) eliminating the need to learn the Makefile specification language, which can be sometimes be complicated, restrictive, and/or error-prone.

We currently have two different versions of tmake: a passive version that brings output up to date in response to a user command, and an active version that automatically updates output files whenever changes to an input file is detected. Both versions can be useful in different contexts. For example, active tmake may be appropriate when a summary graph is produced based on a set of input files. All commands used to create the graph could be re-executed each time a data set changes while a visualization tool detects the changes to the graph and re-displays the current version. In this way, users could interactively manipulate data sets while visualizing the effects on a resulting graph. Passive tmake is likely more appropriate in compilation environments where users change multiple source files and wish to manually instruct tmake to re-synchronize output files.

The passive version of tmake provides an interface similar to traditional `make`. A shell script is used to carry out tasks such as compilation and TREC builds process lineage information. However, passive tmake does not register callbacks on file write events. Rather, the user explicitly requests re-synchronization of the target of the shell script by re-executing the shell script. For each command in the shell script, tmake looks up the set of input files for the command and checks the last modified time of each input files. If any of the files have been modified since the last execution time of the command, the process is re-executed. Otherwise, the command is skipped. Also consider the case where the compile shell script is modified—for example, to add a new `-O` parameter to the compiler. In this case, tmake will re-execute any programs with modified command line parameters even if file dependencies have not changed since it will be unable to match the new process and command line parameters with any entries in its process lineage hierarchy. Thus, passive tmake functions similarly to `make`, while maintaining the advantages of implicitly determining dependency information and dynamically updating dependencies as they change (without requiring user intervention).

With active tmake, the tmake module registers a callback with the TREC tracing module when any file is opened for writing. When the callback is invoked, tmake checks if the file acted as input to any of the traced processes. If so, tmake notifies the user of this update and prompts for re-synchronization of the output files. On a user synchronize command, tmake re-executes the program that took the modified file as input with the same command line parameters and environment variables as the program's initial execution. Once the program completes, tmake recursively checks for further dependencies: if the output files of the just executed program acted as input for any of the program's parents in the lineage tree, the ancestor is in turn re-executed. This process is repeated until an output file is produced that did not act as input to any ancestor in the lineage tree.

To demonstrate the workings of active tmake, consider the process lineage example from Figure 2(b). When the file `test.h` is modified (e.g., through an editor), TREC invokes a callback to the tmake module informing it of the change. Tmake then searches for all processes that used `test.h` as input, finding the cpp process. After asking for confirmation from the user, tmake re-executes cpp, noting that it produced an output file, `/var/tmp.i`. The process is repeated recursively, where tmake notices that the modified file was read by the cc1 process. Processes are executed in this way until ld produces a current version of the `test` executable. Recursion ends at this point since no process took `test` as its input.

Instead of prompting the user for permission to re-synchronize output files, tmake can be configured to skip the prompt and to automatically re-creat output files when any input file is modified. However, such automatic re-synchronization can produce undefined behavior in the general case (e.g., users saving intermediate versions of program source files that will not compile). Of course, earlier work in optimistic make [Bubenik & Zwaenepoel 1989] has demonstrated the value of creating output files in anticipation of user requests. Thus, optimistic versions of output files could be created in temporary directories; once the user requests an update, a new version of the output file can be moved in place of the old one instead of waiting for the file to be re-created.

3.3 Dynamic Web Caching

In this subsection, we describe a third TREC example, dynamic web caching. This service is quite different in motivation and implementation from both the previous services, unmake and tmake. We begin by motivating the need for dynamic web caching and go on to describe how we modified an HTTP server to interact with TREC in order to provide this service.

3.3.1 Motivation

In response to the exponential growth of packets across the Internet, several researchers have proposed a number of caching schemes both to reduce the load on Internet backbones and to improve user response times [Gwertzman & Seltzer 1996, Chankhunthod et al. 1996, Zhang et al. 1997]. One early study [Danzig et al. 1993] found that strategically-placed caches could reduce FTP file traffic by as much as 50%. Similar studies of WWW traffic yielded similar results [Braun & Claffy 1994, Duska et al. 1997, Gribble & Brewer 1997].

We observe that any caching scheme will be limited by the large fraction of web pages that are dynamically generated, and hence classified as uncacheable. For example, a CGI-bin program might be run to produce HTML in response to a user query (e.g., what are the show times at a movie theater) or to embed a different advertisement in the same logical page based on the identity of the requester. In general, the contents of such pages cannot be cached because the result of the program can change from execution to execution. Caching dynamic objects can be even more important for overall performance for two reasons: (i) An increasing percentage of web objects are being dynamically generated, and (ii) web servers typically have to perform `fork` and `exec` operations for dynamic operations, increasing server load and request overhead.

Our approach to reducing the overhead of busy web servers is to cache dynamically generated pages, using TREC to manage invalidations. Limitations to the type of dynamic objects that can be cached—for example, those that access a database—are described in Section 3.3.4. In this scheme, cache objects are stored in the file system under the name of the program used to generate them concatenated with any arguments to the program. Thus, a request for the object `http://www/cgi-bin/query?argument` might be cached locally in, for example, a file `/usr/local/apache/cache/cgi-bin/query?argument`. Subsequent accesses to the same CGI program with the same argument list can be returned from the disk cache, eliminating the need to `fork` and `exec` operations, and saving any computation time associated with the requested program. For example, consider user queries to a web site providing movie show times. Caching is attractive in this context because locality is likely present in the access pattern (popular movie at popular theater) and because the query results remain valid for an extended period of time (e.g., one week).

Of course, one problem with caching dynamic objects is maintaining cache consistency. The dynamically generated web objects often depend on a set of input files. For example, a consumer web site might provide an interface for users to interactively query for the latest pricing and availability information. Dynamic object caching can reduce server load by caching the replies to frequently made requests. However, all cached copies must be invalidated when pricing or availability information changes. As another example, a news site may dynamically generate a "front page" containing headlines and synopsis of news stories. Caching is also useful in this context since the same object will be delivered to all users for a certain time period. Once again, however, cached copies must be invalidated when the list of available stories is updated.

To address this need for invalidation, we use TREC to profile the execution of programs creating dynamic web objects. When TREC detects that an input file contributing to the creation of a cached object has been modified, one of two courses of action can be followed: (i) the file containing the cached copy of the web object is removed, forcing the web server to re-create it on the next user access, or (ii) the program which originally created the cached object can be re-executed to bring the cache up to date. Determining which approach is taken depends on the popularity of the object in question, the current load of the web server, and the cost of recomputing the object.

3.3.2 Implementation

To investigate the utility of dynamic web caching as described above, we modified Apache's HTTP server (version 1.2.4) in the following way. When a CGI object[1] is requested, the server first checks for a file whose name matches the CGI object name concatenated with any arguments. If the file exists, its contents are returned without spawning a new process to carry out the request. If not present, a process is spawned to produce the desired results. The program's output is written to a file in parallel with the response to the requester. Thus, subsequent requests are able to use a cached copy of the CGI object. File locking is used to ensure that partially generated results are not returned to users. We were able to make these changes by modifying approximately 50 lines of C code from the Apache distribution.

To allow for invalidations, the execution of CGI programs is profiled by TREC. Similar to transparent make, the dynamic web caching module registers callbacks for all files that act as input for CGI programs, requesting notification

[1]While our implementation focuses on caching CGI program results, our technique is equally applicable to other dynamically generated web content such as dynamic HTML.

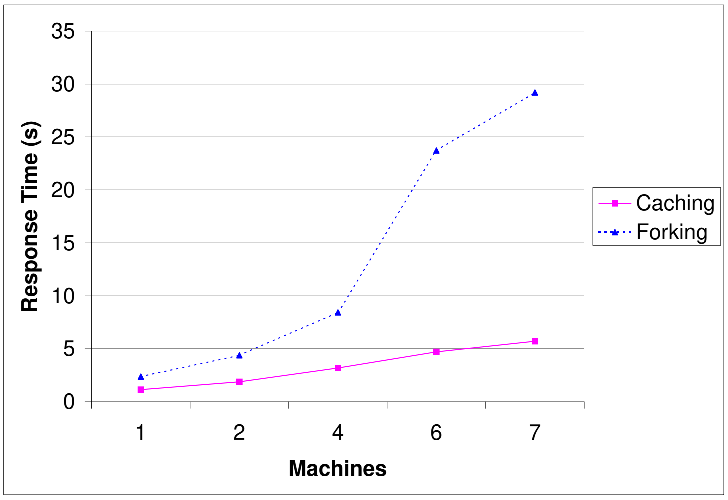


Figure 4: This figure describes the relative performance improvement introduced by CGI object caching.

when any of the target set of files are modified. When such a callback is received, all CGI objects (cached output files) which depended on the modified file are removed, forcing the server to regenerate the result on the next user access. Since this level of dependency checking cannot guarantee consistency for all CGI objects (further discussed in Section 3.3.4), we allow the server administrator to specify a set of CGI programs that cannot be cached through an Apache configuration file.

3.3.3 Performance

To quantify the baseline performance benefits of caching CGI-bin program results, we measured the performance of retrieving CGI results for both our modified Apache server and the original, unmodified version. The measurements were taken as follows. Eight Sun Ultra/1 workstations running Solaris 2.5.1 connected by a 10 Mb/s Ethernet switch were used for the experiments. One of the machines acted as the HTTP server running Apache 1.2.4. Between one and seven of the other machines acted as clients, continuously requesting the results of executing a small CGI script, printenv, which simply prints out the environment variables of the running CGI script. Each client machine forked 40 copies of the same script requesting 200 copies of the same script in a loop. Given the small size of the requests and the replies, network bandwidth was not the bottleneck.

Figure 4 describes the relative performance of the baseline vs. the modified Apache server under the above conditions. As a point of reference, the printenv script takes .05 seconds to run locally. One client requesting the CGI-script in a loop from unmodified Apache averages .18 seconds per request. Using the caching version of Apache, the CGI-script is retrieved in an average of .11 seconds, a 39% improvement over the baseline. From Figure 4, it is interesting to note that the relative performance of the caching CGI server improves with increased load. This improvement results from the high overhead of managing and context switching between address spaces as many CGI scripts are forked off by the baseline HTTP server under load.

We expect the above performance disparity to become even more pronounced as the CGI scripts become longer lived (recall that printenv exits after .05 seconds). To test this hypothesis, we re-ran our experiments with the baseline (uncaching) HTTP server returning the contents of a synthetic CGI program which takes 2.5 seconds to execute. Forty clients on one machine averaged 153 seconds to retrieve the object, while forty clients on each of two machines averaged 167 seconds to retrieve the object. Clearly, the savings from caching become more pronounced as the computation cost of the CGI object becomes more expensive.

3.3.4 Applicability

Our approach to dynamic web object caching faces a number of limitations. First, many dynamic objects are generated as a result of queries full-fledged databases. In essence, many web servers act as simple front ends to a sophisticated DBMS. For example, a user query for a price quote or item availability often translates into a database query. Since access to database tables cannot in general be modeled by simple file accesses (many databases are implemented on top of raw disks as opposed to the file system for example) , TREC cannot catch database table updates, and hence cannot properly invalidate web objects based on out-of-date table values.

While the above limitation is inherent, we believe the performance improvements available from dynamic object caching argues for further research into active databases. For example, research efforts into *materialized views* [Gupta et al. 1993, Gupta & Mumick 1995, Colby et al. 1996, Kawaguchi et al. 1996] in active databases [McCarthy & Dayal 1989, Stonebraker et al. 1990, Widom & Finkelstein 1990] has resulted in support for such views in many commercial database systems. Materialized views allow the results of a query to be updated as the tables (and individual cells) used to create the view are updated. Techniques similar to those employed by TREC are used to track view dependencies on individual cells and tables, and to set "triggers" to be fired when a table is modified so that any derived views can be updated as well. An interesting avenue of future research is to evaluate whether materialized database views can be used to cache the portion of dynamic web objects that generate database queries to obtain their results.

A second limitation faced by dynamic object caching is that, as described, TREC profiling and invalidation only allows for caching on the server-side. If such caching could be extended to Web proxies, performance could be further improved by caching dynamic objects closer to clients, potentially reducing both consumed wide area bandwidth and user-perceived latency. One approach to addressing this limitation is to use a wide-area file system such as AFS [Howard et al. 1988] or WebNFS [Sun Microsystems 1996] to store and to cache dynamic web objects as normal files. Thus, the wide area file system can act as a shared file cache for both the HTTP server and interested proxies, with TREC invalidations maintaining relatively strong consistency semantics. Another approach is to allow proxy caches to cache dynamic objects with a TTL-based invalidation scheme [Chankhunthod et al. 1996, Gwertzman & Seltzer 1996, Squ 1996]. While this approach provides weaker consistency semantics, it is easier to deploy given the current Web infrastructure.

4 Related Work

Several systems have attempted to extend the automatic control of derived objects beyond the simple (but powerful) model used by `make`. DSEE [Leblang & Chase 1984, Leblang & McLean 1985], Odin [Clemm & Osterweil 1990] and Vesta [Levin & McJones 1993, Heydon et al. 1997] provide tools for modeling the behavior of programs, enabling the concise specification of derivation rules, and distributing changes to developers. Their declarative style suits large-scale programming environments, which are highly structured and employ a well-defined set of tools (compilers, linkers, etc.). Neither tool provide any assurance of correctness; as with `make`, the user is responsible for describing the complete set of dependencies relationships to the configuration manager. In contrast to unmake, users of these systems must tell the system how tools use files, whereas unmake simply observes and gathers the information in the background.

Odin relies heavily on the use of naming conventions: the name of a file fully specifies how it was derived. This restriction would not work well for the ad-hoc, highly parameterizable methodology used in less-structured environments. Like tmake, Odin implements transparent re-creation of files. A sentinel in Odin is a data object that is automatically regenerated (if necessary) at the time a user requests it, based on rules that were specified in advance for objects of its type.

VOV [RTDA], a configuration management toolkit, is similar to TREC, in that it observes program invocations to generate a trace of lineage information. However, VOV is limited to a specialized application domain (Electronic CAD), and it requires assistance from tool programmers. Each tool explicitly reports the files it will read and write. By contrast, unmake observes file-system activity at a low enough level that modifying tools to work with TREC is unnecessary.

Recently, a large body of research is being conducted in web caching. Harvest [Chankhunthod et al. 1996] and Squid [Squ 1996] are efforts into hierarchical web proxy caching. We believe that such caching efforts would benefit from our work in dynamic object caching. Gwertzman and Seltzer [Gwertzman & Seltzer 1996] recently proposed using the Alex protocol [Cate 1992] for maintaining cache consistency across the wide area. While this protocol provides much weaker consistency guarantees than a wide area file system, it would be simpler to deploy and could be used in our model for caching dynamic web objects at proxy caches.

5 Conclusions

The task of managing interactions between input and output files can be difficult. Further, the task of manually specifying such dependencies can be tedious and error-prone. We address this problem by introducing Transparent Result Caching (TREC), which automatically and transparently constructs dependency information by observing program behavior. To demonstrate its utility, we have described, built, and evaluated three sample TREC applications. Unmake allows users to query for process lineage information, returning the full chain of processes, command line parameters, and environment variables used to create a file. Transparent Make uses the process lineage information from Unmake to provide functionality similar to UNIX `make`, with the added advantage of freeing users from manually specifying file dependencies. Finally, Dynamic Web Object Caching allows web servers to coherently cache the results of dynamic web content such as CGI programs, with the potential of reducing server load and client latency.

Acknowledgments

Joel Fine did implementation work on an earlier version of transparent make/unmake running on Digital workstations and contributed to an earlier version of this paper. This work greatly benefited from discussions with Paul Eastham, Stefan Savage, Ashutosh Tiwary, and Geoff Voelker.

References

[Apa 1995] *Apache HTTP Server Project*, 1995. `http://www.apache.org/`.

[Bershad & Pinkerton 1988] B. N. Bershad and C. B. Pinkerton. "Watchdogs — Extending the UNIX File System". *Computing Systems*, 1(2):169–188, Spring 1988.

[Braun & Claffy 1994] H.-W. Braun and K. Claffy. "Web Traffic Characterization: An Assessment of the Impact of Caching Documents From NCSA's Web Server". In *Second International World Wide Web Conference*, October 1994.

[Bubenik & Zwaenepoel 1989] R. Bubenik and W. Zwaenepoel. "Performance of Optimistic Make". In *Proceedings of Sigmetrics*, pp. 39–48, 1989.

[Cate 1992] V. Cate. "Alex – a Global Filesystem". In *Proceedings of the 1992 USENIX File System Workshop*, pp. 1–12, May 1992.

[Chankhunthod et al. 1996] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. "A Hierarchical Internet Object Cache". In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[Clemm & Osterweil 1990] G. Clemm and L. Osterweil. "A Mechanism for Environment Integration". *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, Jan 1990.

[Colby et al. 1996] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. "Algorithms for Derred View Maintenance". In *SIGMOD*, pp. 469–480, 1996.

[Danzig et al. 1993] P. B. Danzig, M. F. Schwartz, and R. S. Hall. "A Case for Caching File Objects Inside Internetworks". In *ACM SIGCOMM 93 Conference*, pp. 239–248, September 1993.

[Dozier 1993] J. Dozier. Personal Communication, March 1993.

[Duska et al. 1997] B. Duska, D. Marwood, and M. J. Feeley. "The Measured Access Characteristics of World Wide Web Client Proxy Caches". In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[Ghormley et al. 1996] D. P. Ghormley, D. Petrou, and T. E. Anderson. "SLIC: Secure Loadable Interposition Code". Technical Report CSD-96-920, University of California at Berkeley, November 1996.

[Goldberg et al. 1996] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. "A Secure Environment for Untrusted Helper Applications". In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[Gribble & Brewer 1997] S. D. Gribble and E. A. Brewer. "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace". In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[Gupta & Mumick 1995] A. Gupta and I. S. Mumick. “Maintenance of Materialized Views: Problems, Techniques, and Applications”. In *Data Engineering Bulletin*, June 1995.

[Gupta et al. 1993] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. “Maintaining View Incrementally”. In *SIGMOD*, 1993.

[Gwertzman & Seltzer 1996] J. Gwertzman and M. Seltzer. “World-Wide Web Cache Consistency”. In *Proceedings of the 1996 USENIX Technical Conference*, pp. 141–151, January 1996.

[Heydon et al. 1997] A. Heydon, J. Horning, R. Levin, T. Mann, and Y. Yu. “The Vesta-2 Software Description Language”. Technical Report 1997-005, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1997.

[Howard et al. 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. “Scale and Performance in a Distributed File System”. *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[Jones 1993] M. B. Jones. “Interposition Agents: Transparently Interposing User Code at the System Interface”. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 80–93, December 1993.

[Kawaguchi et al. 1996] A. Kawaguchi, D. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. “Concurrency Control Theory for Deferred Materialized Views”. Unpublished, 1996.

[Leblang & Chase 1984] D. B. Leblang and R. P. Chase, Jr.. “Computer-Aided Software Engineering in a Distributed Workstation Environment”. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 104–112, May 1984.

[Leblang & McLean 1985] D. B. Leblang and G. D. McLean, Jr.. “Configuration management for large-scale software development efforts”. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 122–127, Harwichport, Massachusetts, June 1985.

[Levin & McJones 1993] R. Levin and P. R. McJones. “The Vesta Approach to Precise Configuration of Large Software Systems”. Technical Report 105, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1993.

[McCarthy & Dayal 1989] D. R. McCarthy and U. Dayal. “The Architecure of an Active Data Base Management System”. In *SIGMOD*, June 1989.

[RTDA] “VOV”. `http://www.rtda.com/vov.html`. Runtime Design Automation.

[Squ 1996] *Squid Internet Object Cache*, 1996. `http://squid.nlanr.net/Squid/`.

[Stonebraker et al. 1990] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. “On Rules, Procedures, Caching and Views In Database Systems”. In *SIGMOD*, May 1990.

[Sun Microsystems 1996] “WebNFS: The Filesystem for the World Wide Web”. Technical report, Sun Microsystems, 1996. See `http://www.sun.com/webnfs/wp-webnfs/`.

[Widom & Finkelstein 1990] J. Widom and S. J. Finkelstein. “Set-Oriented Production Rules in Relational Database Systems”. In *SIGMOD*, May 1990.

[Zhang et al. 1997] L. Zhang, S. Floyd, and V. Jacobsen. “Adaptive Web Caching”. In *Web Caching Workshop*. NLANR, June 1997.